SGSI – A Scalable GPU-friendly Subgraph Isomorphism Algorithm

Li Zeng, Lei Zou, M. Tamer Özsu

Abstract—Due to the inherent hardness of subgraph isomorphism, the performance is often a bottleneck in various real-world applications. We address this by designing an efficient subgraph isomorphism algorithm leveraging features of GPU architecture. Existing GPU-based solutions adopt two-step output scheme, performing the same join twice in order to write intermediate results concurrently. They also lack GPU architecture-aware optimizations that allow scaling to large graphs. In this paper, we propose a *S*calable *G*PU-friendly *s*ubgraph *i*somorphism algorithm, *SGSI*. SGSI incorporates a *Prealloc-Combine* strategy based on the vertex-oriented framework, which avoids joining-twice in existing solutions. It uses a GPU-friendly data structure (called *PCSR*) to represent an edge-labeled graph. We also study fine-grained load balance strategies and discuss how to handle enormous graphs that cannot be resident in GPU memory. A partition-based pipeline framework is proposed. Extensive experiments on both synthetic and real graphs show that SGSI outperforms the state-of-the-art algorithms by up to several orders of magnitude and has a good scalability with graph size scaling to billions of edges.

Index Terms—SGSI, GPU, Subgraph Isomorphism, Scalability

1 INTRODUCTION

GRAPHS have become increasingly important in modeling complicated structures and schema-less data such as chemical compounds, social networks and RDF (Resource Description Framework) datasets. The growing popularity of graphs has generated many interesting data management problems. Among these, subgraph search is a fundamental one: how to efficiently enumerate all subgraph isomorphic matches of a query graph over a data graph. This is the focus of the current paper. Subgraph search has many applications, e.g., chemical compound search [1] and search over a knowledge graph [2], [3], [4]. A running example (query graph Q and data graph G) is given in Figure 2 and Figure 2(c) illustrates the matches of Q over G.

Subgraph isomorphism is a well-known *NP-hard* problem [5] and most solutions follow some form of tree search with backtracking [6]. Figure 3 illustrates the search space for Q over G of Figure 2. Although existing algorithms (e.g., [7], [8]) propose many pruning techniques to filter out unpromising search paths, due to the inherent NP-hardness, the search space is still exponential. Therefore, scaling to large graphs with billions of vertices is challenging. One way is to employ hardware assist.

The DFS (depth-first search)-style backtracking tree search is not suitable for parallel computing over GPU. An alternative solution is to employ multiway join to find all matches of Q by leveraging massively parallel processing capability of GPUs to explore the search space in parallel; this follows BFS (Breadth-First Search)-style search. Considering a triangle query Q' with three vertices u_0, u_1 and u_2 (a subgraph component of Q in Figure 2), finding matches of Q' over data graph G is equivalent to multiway join $M(Q') = T_1 \bowtie T_2 \bowtie T_3$, where each T_i contains all data edges matching one query edge in Q[1]. There are two different methods to execute the multiway join: one is to execute a sequence of binary joins to evaluate the query, which is *edge-at-a-time-join*. Figure 1 shows the edge tables T_1 and T_2 matching query edges $\overline{u_0u_1}$ and $\overline{u_1u_2}$ and intermediate join results $T_1 \bowtie T_2$. Note that all existing GPU subgraph isomorphism algorithms follow this edge join strategy [9], [10].

An alternative strategy is worst-case optimal join (WOJ) [11], [12], [13], which matches the query graph one (query) vertex at-a-time using a multiway join operator that performs multiway intersections. We call this *vertex-at-a-time-join* (or simply *vertex-oriented*) approach. Considering the same triangle query $Q'(u_0, u_1, u_2)$, table T_1 in Figure 8 contains all data edges matching query edge $\overline{u_0u_1}$. For each candidate edge $\overline{v_iv_j}$ in T_1, if $\omega = N(v_i, b) \cap N(v_j, b) \cap C(u_2) \neq \phi$, $\{\overline{v_iv_j}\} \times \omega$ generates all triangle query matches containing edge $\overline{v_iv_j}$, where $N(v_i, b)$ denotes neighbors of vertex v_i with edge label b, and $C(u_2)$ denotes all candidate vertices (in G) matching query vertex u_2. Iterating the same computation over all edges $\overline{v_iv_j}$ in T_1 produces all matches of Q.

In this paper we propose the first *vertex-oriented* WOJ-based suite of techniques for subgraph isomorphism computation that is suitable for executing on GPUs for hardware assist. *G*PU-friendly *s*ubgraph *i*somorphism solution (*GSI*) targets in-core processing of graphs that can fit GPU memory, and *S*calable *G*PU-friendly *s*ubgraph *i*somorphism solution (*SGSI*) targets larger graphs for out-of-core processing.

There are two state-of-the-art GPU-based subgraph isomorphism algorithms in the literature: GpSM [9] and Gun-

- *Li Zeng and Lei Zou are from Peking University in China; M. Tamer Özsu is from University of Waterloo in Canada. E-mail: {li.zeng,zoulei}@pku.edu.cn, tamer.ozsu@uwaterloo.ca;*
- *Lei Zou is the corresponding author of this work.*

1. Note that the multiway self-join approaches by default find subgraph homomorphisms, which allow one matched subgraph in the data graph to contain a vertex that matches multiple query graph vertices. In order to find subgraph isomorphism, we require that vertices that are later added through extension cannot repeat the originally matched data graph vertices (in the partial matches). It is a trivial issue that can be easily addressed to avoid duplicate vertices in one match.

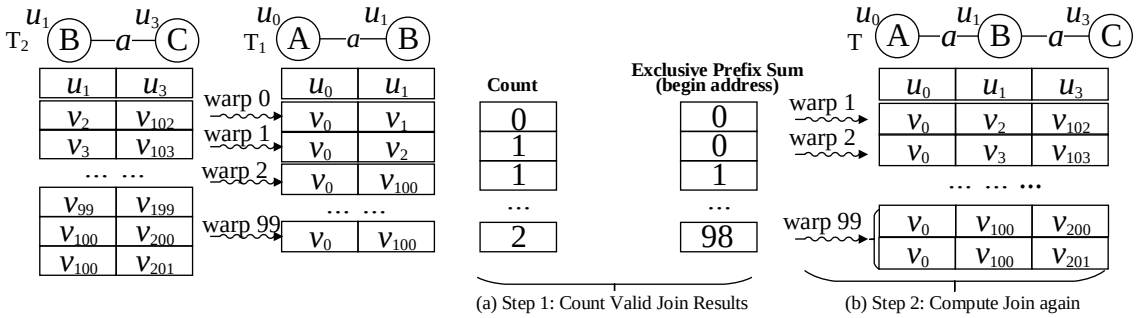


Fig. 1. An example of "two-step output scheme"

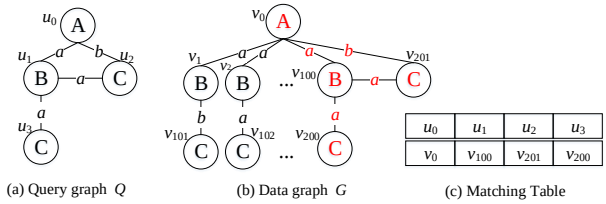


Fig. 2. An example of Query Graph and Data Graph

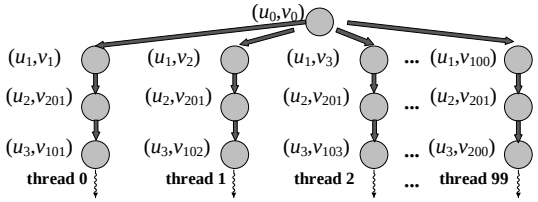


Fig. 3. An example of search tree of Q in G

rockSM [10], both of which are edge-at-a-time-join strategies. They first collect candidates for each edge of a query graph and perform a sequence of binary joins over edge tables. The edge-at-a-time-join strategy suffers from high overhead when implemented on a GPU. A key issue is how to write join results to GPU memory in a massively parallel manner. GpSM and GunrockSM employ the "join-twice output scheme" [14], as illustrated in Example 1.

Example 1 *Consider Q and G in Figure 2. Tables T_1 and T_2 (in Figure 1) show the edges matching $\overline{u_0u_1}$ and $\overline{u_1u_3}$, respectively. In order to obtain matches of the subgraph induced by vertices u_0, u_1 and u_3, GpSM performs the edge join $T_1 \bowtie T_2$. Assume that each processor handles one row in T_1 for joining. Writing the join results to memory in parallel may lead to a conflict, since different processors may write to the same address.*

To avoid this, the naive solution is locking, but that reduces parallelism. GpSM and GunrockSM instead use "join-twice output scheme". In the first step, each processor joins one row in T_1 with the entire table T_2 and counts valid matches (Figure 1(a)). Then, based on the prefix-sum, the output addresses for each processor are calculated. In the second step, each processor performs the same join again and writes the join results to the calculated memory address in parallel (Figure 1(b)).

The join-twice output scheme doubles the amount of join work, thus suffers performance issues when GPU is short of threads on large graphs. To avoid joining twice, GSI joins, at each iteration, the intermediate result table M (i.e., the partial matches) with a candidate vertex set (i.e., the candidates matching the query vertex to be joined). To write the join results to memory in parallel, it pre-allocates enough memory space for each row of M and performs the vertex join only once. GSI uses vertex rather than edge as the basic join unit, because it is hard to estimate memory space for edge join results, but it is easy for vertex join. More details are given in Section 5.

Vertex join has two important primitive operations: accessing one vertex's neighbors based on edge labels, and set operations. To support the two primitives efficiently, we propose an efficient data structure called *PCSR* (Section 4), and an efficient GPU algorithm for set operations (Section 5). Specifically, PCSR accelerates the retrieval of a vertex's neighbors with specific edge labels in GPU, and we propose a multi-granularity GPU-based set intersection. We also propose a histogram-based strategy to achieve both inter-block and intra-block load balance.

Many real-world graphs are too large to be resident in GPU memory, requiring an out-of-core solution. PBE [15] is the only existing proposal for out-of-core subgraph search over large graphs. It divides a large graph G into a set of partitions (subgraphs) using an edge-cut strategy (using METIS partitioning algorithm [16]). It then uses different methods to search *intra-partition matches* and *inter-partition matches*. In order to find inter-partition matches, PBE begins the search process from each crossing edge between different partitions. To extend inter-partition partial matches, the required adjacency lists are fetched from the main memory (host memory) and transferred into GPU memory. Thus, PBE uses CPU to fetch the required adjacency lists and organize them into a continuous array so that they can be transferred to GPU. Obviously, this is an expensive operation on the CPU side (see more detailed discussion in Section 2.4).

In contrast, SGSI adopts the multi-way WOJ strategy, which follows a breadth-first-search (BFS) search paradigm. At each join step, SGSI always extends all current size-k partial matches to size-$k+1$ ones. Different from PBE, SGSI does not distinguish *intra-partition* matches and *inter-partition* matches; all matches have the same search strategy. To deal with large graphs out of GPU memory, SGSI needs to move data from main memory to GPU, for which we propose a series of optimizations as discussed in 6.

Both of our methods – GSI for in-core and SGSI for out-of core – have the same search strategy for subgraph search based on WOJ. The only difference is that SGSI needs to read required neighbor lists from main memory and partitioning the intermediate results if they are too large to be resident at GPU memory.

The contributions of this paper are the following:

- We propose an efficient data structure (PCSR) to represent edge-labeled graphs, which helps reduce memory latency in GPU-based subgraph isomorphism algorithm.
- Using *vertex-oriented* join, we propose Prealloc-Combine instead of two-step output scheme, which is significantly more efficient. Leveraging GPU features, we discuss ef-

ficient implementation of set operations, as well as optimizations including load balance and duplicate removal.
- We study several strategies to deal with large graphs that is beyond of the capacity of GPU memory, including the partition and the processing framework.
- We conduct experiments on both synthetic and real large graph datasets that show that GSI outperforms the state-of-the-art approaches (both CPU-based and GPU-based) by several orders of magnitude. Also, our method has good scalability with graph size scaling to billions of edges.

2 Preliminaries & Related Work

We first briefly review the terminology that we use in this paper. Table 1 lists the symbols that are used.

TABLE 1
Notations

G, Q	Data graph and query graph, respectively
v, u	Vertex in G and Q, respectively
$S(v), S(u)$	Encoding of vertex v or u
$N(v), N(u)$	All neighbors of vertex v or u
$N(v, l)$	Neighbors of vertex v with edge label l
$freq(l)$	Frequency of label l in G
$C(u)$	The candidate set of query vertex u in G
M,M'	The old and new intermediate result table, each row represents a partial answer, each column corresponds to a query variable
$num(L)$	The number of currently valid elements in set L
$\|A\|$	The size of set A
$D = P(G, l)$	Edge label l-partitioned subgraph of G

2.1 Problem Definition

Definition 1 ***(Graph)*** *A graph is denoted as $G = \{V, E, L_V, L_E\}$, where V is a set of vertices; $E \subseteq V \times V$ is a set of undirected edges in G; L_V and L_E are two functions that assign labels for each vertex in $V(G)$ and each edge in $E(G)$, respectively.*

Definition 2 ***(Subgraph)*** *A graph $G' = \{V', E', L'_V, L'_E\}$ is a* subgraph *of graph $G = \{V, E, L_V, L_E\}$ if and only if $V' \subseteq V$ and $E' \subseteq E$. The $\subseteq$ operation checks corresponding labels as well.*

Definition 3 ***(Graph Isomorphism)*** *Given two graphs H and G, H is* isomorphic *to G if and only if there exists a bijective function f between the vertex sets of G and H (denoted as $f : V(H) \longrightarrow V(G)$), such that*

- *$\forall u \in V(H), f(u) \in V(G)$ and $L_V(u) = L_V(f(u))$, where $V(H)$ and $V(G)$ denote all vertices in graphs H and G, respectively.*
- *$\forall \overline{u_i u_j} \in E(H)$, $\overline{f(u_i) f(u_j)} \in E(G)$ and $L_E(\overline{f(u_i) f(u_j)}) = L_E(\overline{u_i u_j})$, where $E(H)$ and $E(G)$ denote all edges in graphs H and G, respectively.*
- *$\forall \overline{u_i u_j} \in E(G)$, $\overline{f^{-1}(u_i) f^{-1}(u_j)} \in E(H)$ and $L_E(\overline{f^{-1}(u_i) f^{-1}(u_j)}) = L_E(\overline{u_i u_j})$*

Definition 4 ***(Subgraph Isomorphism Search)*** *Given query graph Q and data graph G, the subgraph isomorphism search problem is to find all* subgraphs *G' of G such that G' is* isomorphic *to Q. G' is called a* match *of Q.*

This paper proposes an efficient GPU-based solution for subgraph isomorphism search. Without loss of generality, we assume Q is connected and use v, u, $N(v)$, $N(v, l)$, $num(L)$, and $|A|$ to denote a data vertex, a query vertex, all neighbors of v, all neighbors of v with edge label l, i.e., $\{v' | \overline{vv'} \in E(G) \wedge L_E(\overline{vv'}) = l\}$, the number of currently valid elements in set L, and the size of set A, respectively. Note that our method can also support other graph pattern matching semantics, such as homomorphism and edge isomorphism; and can process multi-labeled graphs as well. We give the details in the supplementary material of this paper.

2.2 GPU Architecture

GPU is a discrete device that contains dozens of streaming multiprocessors (SM) and its own memory hierarchy. Each SM contains hundreds of cores and CUDA (Compute Unified Device Architecture) programming model provides several thread mapping abstractions, i.e., a thread hierarchy.

Thread Hierarchy. Each core is mapped to a *thread* and a *warp* contains 32 consecutive threads running in Single Instruction Multiple Data (SIMD) fashion. When a warp executes a branch, it has to wait though only a portion of the threads take a particular branch; this is termed as *warp divergence*. A *block* consists of several consecutive warps and each block (having at most 1024 threads) resides in one SM. Each process launched on GPU (called a *kernel function*) occupies a unique *grid*, which includes several equal-sized blocks.

Memory Hierarchy. In Figure 4, *global memory* is the slowest and largest layer. Each SM owns a private programmable high-speed cache, *shared memory*, that is accessible by all threads in one block. Although the size of shared memory is quite limited (Taking Titan XP as example, only 48KB per SM), accessing shared memory is nearly as fast as thread-private registers. Access to global memory is done through 128B-size transactions and the latency of each transaction is hundreds of times longer than access to shared memory. If threads in a warp access the global memory in a consecutive and aligned manner, fewer transactions are needed. For example, only 1 transaction is used in coalesced memory access (Figure 5) as opposed to 3 in uncoalesced memory access (Figure 6).

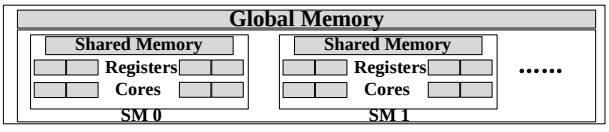


Fig. 4. Memory Hierarchy of GPU

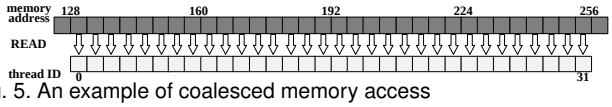


Fig. 5. An example of coalesced memory access

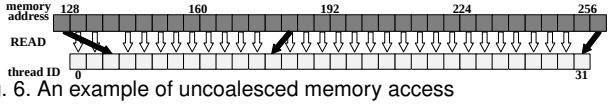


Fig. 6. An example of uncoalesced memory access

2.3 Challenges of GPU-based Subgraph Isomorphism

Although GPU is massively parallel, a naive use of GPU may yield worse performance than highly-tuned CPU algorithms. There are three challenges in designing GPU algorithms for subgraph isomorphism that we discuss below.

Amount of Work. Let $|V(G)|$ and $|V(Q)|$ be the number of

vertices of G and Q, the amount of work is $|V(G)|^{|V(Q)|}$ in Figure 3. If there are sufficient number of threads, all paths can be fully parallelized. But that is not always possible and too much redundant work will degrade the performance. GpSM's strategy (filtering candidates and joining them) is better as it prunes invalid matches early. However, Example 1 shows that the join-twice output scheme used in GpSM doubles the amount of work in join processing, which is a key issue that must be overcome.

Memory Latency. Large graphs can only be placed in global memory. In subgraph isomorphism, we need to perform $N(v, l)$ extractions many times, and they are totally scattered due to inherent irregularity of graphs [17]. It is hard to coalesce memory access in this case, which aggravates latency.

Load Imbalance. GPU performs best when each processor is assigned the same amount of work. However, neighbor lists vary sharply in size, causing severe imbalance between blocks, warps and threads. Balanced workload is better, because the overall performance is limited by the longest workload.

2.4 Related Work

CPU-based subgraph isomorphism. Reference [18] and VF2 [19] are the two early efforts. Reference [18] uses the depth-first search strategy, while VF2 proposes several graph topology-oriented pruning rules. Most later methods [20], [21], [22], [23] pre-compute some structural indices to reduce the search space and optimize the matching vertex order using various heuristic methods. They build indices based on vertex labels and neighborhood structures. One problem in existing methods is the super-linear space complexity of the index structure. Lee et al. [24] experimentally compare some of the above methods and also propose TurboISO [25], which merges similar query vertices and enumerates all paths to find the best matching order. Turbo$_{HOM++}$ [7] further extends Turbo$_{\text{ISO}}$ to handle SPARQL queries over RDF graphs. BoostISO [26] extends the concept of neighborhood equivalence class to data graphs and defines four types of vertex relationships to further reduce duplicate computation. gStore [27] and its enhanced version [4] use the idea of vertex neighborhood encoding to find candidates, while Nauty [28] pre-computes all automorphisms within a data graph to reduce the cost of subgraph isomorphism. CFL-Match [8] defines a Core-Forest-Leaf decomposition and selects the matching order based on minimal growth of the intermediate table. VF3 [29] improves VF2 by using more pruning rules (node classification, matching order, etc.) and favors dense queries.

The above methods follow depth-first search with backtracking, but they differentiate in search ordering, pruning techniques and pre-compute costs. Different from these, EmptyHeaded [12] and CBWJ [13] aim to solve subgraph isomorphism based on edge table join, following a breath-first search. They combine worst-case optimal join [30] and binary join to generate better plans. CBWJ explores larger plan space and adopts more precise cost estimator, achieving better performance than EmptyHeaded.

The recent experimental work [31] provides the comprehensive comparison with different subgraph isomorphism algorithms. However, due to inherent exponential search space, the scalability and performance are two major concerns when running CPU-based algorithms on large data graphs.

GPU-based subgraph isomorphism. Computing subgraph isomorphism on GPUs allows the exploitation of the massive parallelism of these devices. The first work along this line is Sun and Luo [32], which first finds candidates for STwigs [33] and then joins these intermediate results. It uses hash method instead of binary search to join two relation tables of STwigs, indicating a constant time cost. However, the hash method always brings the heavy cost of random memory access, which limits its performance. STwig-based framework is not suitable for GPU due to large intermediate results.

GPUSI [34] adapts TurboISO to GPU by in parallel search of different candidate regions. Its performance is limited by depth-first search within each region. Backtracking-based GPU algorithms always have issues of warp divergence and uncoalesced memory access [35], leading to bad performance.

GpSM [9] and GunrockSM [10] (based on [36]) outperform previous works by leveraging *breadth-first search* that favors parallelism in GPU. Their main ideas have been introduced in Section 1. To avoid parallel write conflicts, they both adopt join-twice output scheme to write join results, doubling the workload.

MAGiQ [37] and Wukong+G [38] are two GPU-based RDF systems that support SPARQL queries. Wukong+G develops an efficient swapping mechanism between CPU and GPU, while MAGiQ utilizes existing CUDA libraries of linear algebra for filtering. Wukong+G is based on relational table join, processing one triple at a time, and develop an efficient swapping mechanism between CPU and GPU. MAGiQ is based on linear algebra and utilizes existing CUDA libraries to perform matrix operations. It finds candidates for all query edges and joins these tables one-by-one, with all non-tree edges being verified at the end. These two systems are not optimized for table joins, making them inefficient in some large data graphs.

Recently, Wang and Owens [39] propose a subgraph matching algorithm based on Gunrock [40], but it mainly targets unlabeled graphs. This is different from the design goal of our proposed PCSR data structure (Section 4) that focuses on edge-labelled graphs. Furthermore, it adopts 2-look-head neighborhood filtering [29], which only fits for induced subgraph isomorphism rather than general cases of subgraph isomorphism.

To support a large data graph that is beyond the capability of GPU memory, the recent work PBE [15] divides the graph into several paritions each of which can be placed in GPU memory. It processes one partition at-a-time, searching all matches within the partition since it is a small graph. To find matches across several partitions, PBE enumerates each query edge e as the cross-partition edge, then expands the solution from matches of e. Furthermore, it proposes a shared execution strategy to eliminate redundant matches. In this way, PBE can scale to graphs with billions of edges. However, the search of cross-matches is very costly. PBE needs to collect all neighbor sets of the intermediate table and organize them as a big array, which is done by CPU and is a long and tedious work. PBE assumes that neighbor sets in this array and intermediate results can be stored on GPU memory, which limits its scalability over large graphs.

Graph mining. GraphZero [41], Peregrine [42] and Sandslash [43] are all single-machine graph mining systems, which provide high-level interface for productivity at the cost of expressiveness and performance. They are CPU-based, and cannot be easily ported to GPU; they cannot scale to large graphs either. Pangolin [44] supports both CPU and GPU processing and is mainly designed for the graph mining tasks, including frequent subgraph mining, subgraph matching and clique detection. However, Pangolin is designed for *unlabelled* graphs and focus on small-size pattern graphs (no more than 4 edges). This is different from

our problem definition. One of our contributions is to design an efficient GPU-oriented data structure (i.e.,*PCSR* in Section 4) for *edge-labelled* graphs.

3 Solution Overview

Our solution consists of two phases: filtering and joining. In the filtering phase, a set of candidate vertices in data graph G are collected for each vertex in the query graph Q; in the joining phase, these candidate sets are joined according to the constraints of subgraph isomorphism (see Definition 4). We discuss how to use GPU to accelerate both phases in the following sections.

3.1 Filtering Phase

Generally, a lightweight filtering method with high pruning power is desirable. Many pruning techniques have been proposed (e.g., [4]). The basic pruning strategy, for vertex-oriented approach, is based on "neighborhood structure-preservation": if vertex v in G matches vertex u in Q, the neighborhood structure around u should be preserved in the neighborhood around v. In this work, we propose a suitable data structure that fits GPU architecture to implement pruning.

We encode the neighborhood structure around a vertex v in G as a length-N bit vector signature $S(v)$ with two parts. The first part is vertex label encoding that hashes a vertex label into K bits. The second part encodes the adjacent edge labels together with the corresponding neighbor vertex. We divide the $(N-K)$ bits into $\frac{N-K}{2}$ groups with two bits per group. For each (edge, neighbor) pair (e, v') of a vertex v, we combine $L_E(e)$ and $L_V(v')$ (i.e., the labels of edge e and v') into a string key and hash it to a bounded integer (MurmurHash[2] is used in our implementation), which represents the ID of some group. Each group has three states: "00"– no pair is hashed to this group; "01"–only a single pair is hashed to this group; and "11"–more than one pair are hashed to this group. Figure 7(a) illustrates vertex signature $S(v_0)$ of G in Figure 2. We compute offline all vertex signatures in G and record them in a signature table (see Figure 7(b)). We have the same encoding strategy for each vertex u in Q. It is easy to prove that if $S(v)\&S(u) \neq S(u)$, v is definitely not a candidate for u ("&" means "bitwise AND operation"). We discuss the parameter tuning in Appendix C in the supplementary materials.

Given a query graph Q, we compute online vertex signatures for Q. For each query vertex u, we check all vertex signatures in the table (such as Figure 7(b)) to fix candidates. We can perform filtering in a massively parallel fashion. Furthermore, the natural load balance of accessing fixed-length signatures is suitable for GPU. To further improve the performance, we organize the vertex signature table column-first instead of row-first. Recall that all threads in a warp read the first element of different signatures in the table, the row-first layout leads to gaps between memory accesses (see Figure 7(c)), i.e., these memory accesses cannot be coalesced. Instead, the column-first layout provides opportunities to coalesce memory accesses (see Figure 7(d)).

3.2 Joining Phase

The outcome of filtering are candidate sets for all query vertices. For the example in Figure 2, candidate sets are $C(u_0) = \{v_0\}$, $C(u_1) = \{v_1, v_2, ..., v_{100}\}$, and $C(u_2) = C(u_3) = \{v_{101}, v_{102}, ..., v_{201}\}$. Figure 8 demonstrates our vertex-oriented join strategy. Assume that we have matches of edge $\overline{u_0u_1}$ in table M and candidate vertices $C(u_2)$. In Q, u_2 is linked to u_0 and u_1 according to the edge labels b and a, respectively. Thus, for each record (v_i, v_j) in M, we read $N(v_i, b)$ and $N(v_j, a)$ and do the set operation $N(v_i, b) \cap N(v_j, a) \cap C(u_2) \setminus \{v_i, v_j\}$, where $N(v_i, b)$ and $N(v_j, a)$ denote neighbors of v_i with edge label b and v_j with edge label a, respectively. If the result is not empty, new partial answers can be generated, as shown in Figure 8.

Notice that there are two primitive operations: accessing one vertex's neighbors based on the edge label (i.e., $N(v, l)$ extraction) and set operations. We use a novel data structure for graph storage on GPU (Section 4) that aids this access. We have efficient implementation of set operations, including a parallel join algorithm (Section 5).

4 Data Structure of Graph: PCSR

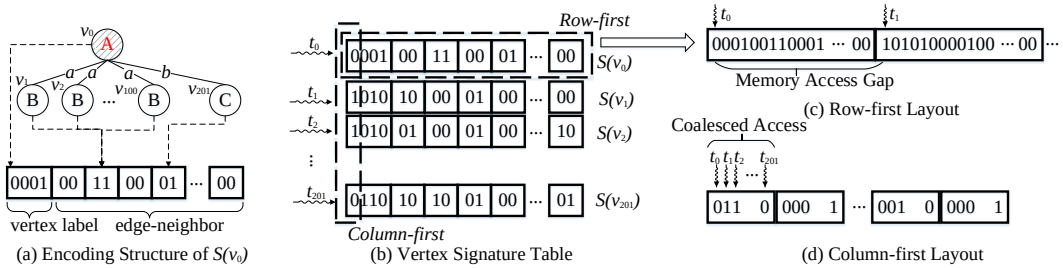
Compressed Sparse Row (CSR) [45] is widely used in existing algorithms (e.g., GunrockSM and GpSM) over sparse matrices or graphs, and it allows locating one vertex's neighbors in $O(1)$ time. Figure 9 shows, as an example, the 3-layer CSR structure of G in Figure 2. The first layer is "row offset" array, recording the address of each vertex's neighbors. The second layer is "column index" array, which stores all neighbor sets consecutively. The corresponding weight/label of each edge is stored in "edge value" array. If no edge weight/label exists, we can remove "edge value" array and work with the 2-layer CSR structure. To extract $N(v, l)$ in CSR, all neighbors of v must be accessed and checked whether or not corresponding edge label is l. Obviously, the memory access latency is very high and it suffers from severe thread underutilization because threads extracting wrong labels are inactive thus wasted. We carefully design a GPU-friendly CSR variant to support accessing $N(v, l)$ efficiently. The complexity of $N(v, l)$ extraction consists of locating and enumerating. In our structures, $N(v, l)$ is stored consecutively, i.e., the complexity of enumerating is the same: $O(|N(v, l)|)$. Thus, we use the time complexity of locating $N(v, l)$ as a metric.

To speed up memory access, we divide G into different *edge label-partitioned graphs* (for each edge label l, the edge l-partitioned $P(G, l)$ is the subgraph G' (of G) induced by all edges with label l). These partitioned graphs are stored independently and edge labels are removed. The straightforward way is to store each one using traditional CSR. However, this would not work well, since vertex IDs in a partitioned graph are not consecutive. For example, the edge partitioned graph $P(G, b)$ has only two edges and four vertices (v_0,v_1,v_{101},v_{201}). The non-consecutive vertex IDs disable accessing the corresponding vertex in the row offset in $O(1)$ time (by vertex ID). There are two simple solutions:

(1) *Basic Representation*. The entire vertex set $V(G)$ is maintained in the row offset for each edge partitioned graph CSR, regardless of whether or not a vertex v is in the partitioned graph (see Figure 10(a)). Clearly, this approach can locate a vertex's neighbors in $O(1)$ time using the vertex ID directly, but it has high space cost: $O(|E(G)| + |L_{E(G)}| \times |V(G)|)$, where $|L_{E(G)}|$ is the number of distinct edge labels. In complex graphs such as DBpedia, there are tens of thousands of different edge labels and this solution is not scalable.

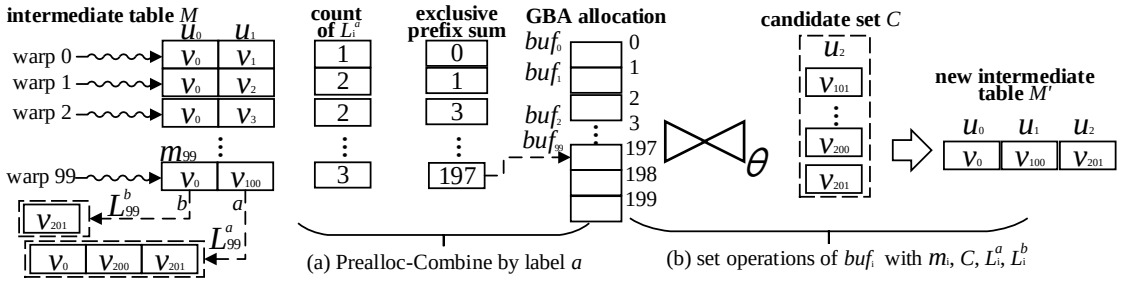
(2) *Compressed Representation*. A layer called "vertex ID" is added, and binary search is performed over this layer to find

2. MurmurHash is a non-cryptographic hash function suitable for general hash-based lookup. More details can be found at https://en.wikipedia.org/wiki/MurmurHash



(a) Encoding Structure of $S(v_0)$ (b) Vertex Signature Table (c) Row-first Layout (d) Column-first Layout

Fig. 7. Encoding table of data vertices



(a) Prealloc-Combine by label a (b) set operations of buf_i with m_i, C, L_i^a, L_i^b

Fig. 8. Vertex-oriented Join Strategy

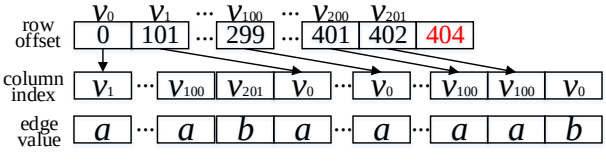


Fig. 9. Traditional CSR structure

corresponding offset (see Figure 10(b)). Obviously, the overall space cost is lowered, which can be formulated as $O(|E(G)|)$. However, this leads to more memory latency. Theoretically, we require $\lceil \log(|V(G,l)|+1) \rceil + 2$ memory accesses to locate $N(v,l)$, where $|V(G,l)|$ denotes the number of vertices in the edge l-partitioned graph $P(G,l)$.

Therefore, neither of the above methods work for a large data graph G. In the following we propose a new GPU-friendly data structure to access $N(v,l)$ efficiently, called *PCSR* (Definition 5). We reorganize the row offset layer using hashing. The row offset layer is an array of hash buckets, called *group*. Each item hashed to the group is a pair (v, o_v), where v is a vertex ID and o_v is the offset of v's neighbors in column index ci. Let Θ be a constant to denote the maximum number of pairs in each group. The last pair is an *end flag* to deal with the overflow. We require that $2 \leq \Theta \leq 16$, then one group can be read concurrently by a *single* memory transaction using one warp.

Definition 5 ***PCSR structure.*** *The Partitioned Compressed Sparse Row of an edge l-partitioned graph $P(G,l)$ is defined as follows:*

- *ci is the column index layer that holds the neighbors.*
- *$gl = \{g_i\}$ is an array of groups and each group is a collection of pairs. Each group has no more than Θ pairs.*
- *Each pair in g_i is denoted as (v, o_v) except for the last pair, where v is a vertex ID and o_v is the offset of v's neighbors in the column index ci, i.e., a prefix sum of the number of neighbors for vertices. Let n_v be the offset of next pair. v's neighbors start at $ci[o_v]$ and end before $ci[n_v]$. All vertices in one group have the same hash value.*
- *The last pair (GID, END) is the overflow flag. $GID = -1$ means no overflow; otherwise, overflowed vertices are stored in the GID-th group. Note that $g_i.END$ is the end position of previous vertex's neighbors in ci, i.e., the first o_v in group g_{i+1}.*

Figure 10(c) is an example of PCSR corresponding to edge a-partitioned graph. Let D denote $P(G,l)$, the edge label l-partitioned graph. Algorithm 7 in Appendix A builds PCSR for D. We hash $|V(D)|$ vertices into Δ groups. For each node v, we hash v to one group using a hash function f (Lines 3-4). If some group g_i overflows (i.e., more than $\Theta - 1$ vertices are hashed to this group), we find another empty group g_j (we can tune parameter Δ to guarantee that we can always find empty groups to store these overflowed vertices and more details are given later) and record group ID of g_j in the last pair in g_i to form a linked list (Lines 5-8).

Based on PCSR, we compute one vertex's neighbors according to edge label. An example of computing $N(v_0, a)$ in Figure 10(c) is given as follows.

1) use the same hash function f to compute the group ID idx that v_0 maps to, here $idx = 0$;
2) read the entire 0-th group (i.e., g_0) to shared memory concurrently using one warp in one memory transaction;
3) probe all pairs $(v', o_{v'})$ in this group (g_0) concurrently using one warp;
4) we find the first pair (in group g_0) that contains v_0. The corresponding offset is 0 and the next offset 100. It means that $ci[0, ..., 99]$ in the column index layer are v_0's neighbors based on edge label a.

Assume that vertex v is hashed to the i-th group g_i. Due to the hash conflict, v may not be in group g_i. In this case, according to the last pair, we can read another group whose ID is $g_i.GID$ and then try to find v in that group. We iterate the above steps until v is found in some group or a group is found whose $g_i.GID$ is "-1" (i.e., v does not exist in the edge labelled partitioned graph D). The time and space cost of PCSR are analyzed in Appendix A.

Parameter Setting Δ: In our implementation, $\Delta = |V(D)|$,

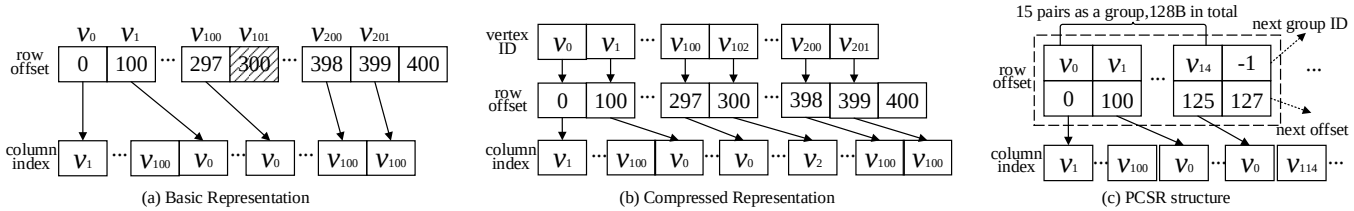


Fig. 10. Three Representations of edge a-partitioned graph

namely, we always hash $|V(D)|$ vertices into $|V(D)|$ groups. This is done for two reasons. First, when $\Delta = |V(D)|$, once some group g_i overflows, we can always find empty groups to store the overflow vertices (Claim 1 of Appendix A proves this). Second, under $\Delta = |V(D)|$, the expected number of memory transactions for reading $N(v, l)$, neighbors of v with edge label l is amortized $O(1)$ (see *Time/Space Complexity* of Appendix A). Obviously, $\Delta > |V(D)|$ leads to less hash conflict, but sacrifices more GPU memory. Therefore, we set $\Delta = |V(D)|$.

5 Parallel Join Algorithm

In the following discussion, we assume that the data graph G can be resident in GPU's global memory. In Section 6, we extend our approach to deal with large graphs that are beyond of capacity of GPU's global memory.

Algorithm 1 outlines the join algorithm, where the intermediate table M stores all matches of partial query Q'. In each iteration, we consider one query vertex u and join M with candidate set $C(u)$ (Lines 9-11). Heuristically, the first selected vertex u' has the minimum $score(u') = \frac{C(u')}{deg(u')}$ (Lines 5-7). In later iterations, we consider the adjacent edge label frequency ($freq(l)$) when selecting the next query vertex to be joined.

Algorithm 1: The whole join process

```
   Input: query graph Q, data graph G
   Output: the final matches of Q in G
1  Let Q' be the partial query graph, set Q' = φ;
2  foreach node u' in Q do
3      score(u') = C(u')/deg(u');
4  for i = 1 to |V(Q)| do
5      if i == 1 then
6          u_c = argmin_{u'} score(u');
7          set intermediate table M = C(u_c) and add u_c to Q';
8      else
9          u = argmin_{u' ∉ Q'} {score(u') | u' is connected to Q'};
10         Call Algorithm 2 to join M with C(u) (generating
             new intermediate table M');
11         set M = M', u_c = u and add u to Q';
12     foreach edge u_c u' in Q do
13         score(u') = score(u') × freq(L_E(u_c u'));
14 return M as final result;
```

Algorithm 2 lists how to process each join iteration (i.e., Line 10 in Algorithm 1). We first study some of its key components. Each warp in GPU joins one row of M with candidate set $C(u)$: acquires neighbors of vertices in this row leveraging restrictions on edge labels, and intersects them with $C(u)$ (*set intersection*). The result of the intersection should remove the vertices in this row (*set subtraction*), to satisfy the definition of isomorphism.

Let Q' be the partial query graph induced by query vertices u_0 and u_1. Figure 8 shows the intermediate table M, in which each row m_i represents a partial match of Q'. For each row m_i, we assign a buffer (buf_i) to store temporary results. Assuming that the next query vertex to be joined is u_2, let us consider the last warp w_{99} that deals with the last row $m_{99} = \{v_0, v_{100}\}$. There are two linking edges $\overline{u_0u_2}$ and $\overline{u_1u_2}$ with edge labels a and b, respectively. Warp w_{99} works as follows:

1) Read v_0's neighbors with edge label a, i.e., $N(v_0, a)$;
2) Write $buf_{99} = (N(v_0, a) \setminus \{v_0, v_{100}\}) \wedge C(u_2)$;
3) Read v_{100}'s neighbors with edge label b, i.e., $N(v_{100}, b)$;
4) Update $buf_{99} = buf_{99} \wedge N(v_{100}, b)$.
5) If $buf_{99} \neq \phi$, each item in buf_{99} can be concatenated to the partial match m_{99} to form a new match of $Q' \cup u_2$. We write these matches to a new intermediate table M.

Problem of Parallelism. When all warps write their corresponding results to global memory concurrently, conflicts may occur. To manage concurrent writing, existing solutions use *join-twice output scheme*, which means the join is performed twice. In the first round, the valid join results for each warp are counted. Based on prefix-sum of these counts, each warp is assigned an offset. In the second round, the join process is repeated and join results are written to the corresponding addresses based on the allocated offsets. An example has been discussed in Example 1. Obviously, this approach doubles the amount of work.

Prealloc-Combine. Our method, called "Prealloc-Combine", performs the join only once (Algorithm 2). Each warp w_i joins one row (m_i) in M with candidate set $C(u)$. Before join processing, for each warp w_i, we allocate memory for buf_i to store all valid vertices that can be joined with row m_i (Line 1 in Algorithm 2). A question is how to decide the size of each buffer buf_i. Let Q' be the partial query graph that has been matched. We select one linking edge $e_0 = \overline{u'_0u}$ in query graph Q ($u'_0 \in V(Q')$), and u ($\notin V(Q')$) is the query vertex to be joined. Assume that the edge label is "l_0". As noted above, m_i denotes one partial match of query graph Q'. Assume that vertex v'_i matches u'_0 in m_i. It is easy to prove that the capacity of buf_i is upper bounded by the size of $N(v'_i, l_0)$. Based on this observation, we can pre-allocate memory of size $|N(v'_i, l_0)|$ for each row. Note that this pre-allocation strategy can only work for "vertex-oriented" join, since we cannot estimate the join result size for each row in the "edge-oriented" strategy. During each iteration, the selected edge e_0 is called *the first edge* and it should be considered first in Line 2 of Algorithm 2. For example, in Figure 8, $\overline{u_1u_2}$ is selected as e_0, thus the allocated size of buf_{99} should be $|N(v_{100}, a)| = 3$.

Though buffers can be pre-allocated separately for each row (i.e., each row issues a new memory allocation request), it is better to combine all buffers into a big array and assign consecutive memory space (denoted as GBA) for them (only one memory allocation request needed). Each warp only needs to record the offset within GBA. The benefits are two-fold:

(1) *Space Cost*. Memory is organized as pages and some pages may contain a small amount of data. In addition, pointers to buf_i

need an array for storage (each pointer needs 8B). Combining buffers together helps reduce the space cost because it does not waste pages and only needs to record one pointer (8B) and an offset array (each offset only needs 4B).

(2) *Time Cost*. Combined preallocation has lower time overhead due to the reduced number of memory allocation requests. Furthermore, the single pointer of GBA can be well cached and the number of global memory load transactions decreases thanks to the reduction in the space cost of pointer array.

Algorithm 2: Join a new candidate set

Input: query graph Q, current intermediate table M' corresponding to the partial matched query Q', candidate set $C(u)$ (u is the vertex to be joined), and linking edges ES between Q' and u.
Output: new intermediate table M

1 Call Algorithm 3 to select the first edge e_0, and pre-allocate memory GBA and offset array F.
2 **foreach** *linking edge* $e = \overline{u'u}$ *in* ES **do**
3 let l be the label of edge e in Q;
4 launch a GPU kernel function to join M' with $C(u)$;
5 **forall** *each row* m_i *(partial match) in* M **do**
6 let buf_i be the segment F_i~F_{i+1} in GBA;
7 assign a unique warp w_i to deal with m_i;
8 assume that v'_i match u' in m_i;
9 **if** e *is the first edge* e_0 **then**
10 do set subtraction $buf_i = N(v'_i, l) \setminus m_i$;
11 do set intersection $buf_i = buf_i \cap C(u)$;
12 **else**
13 do set intersection $buf_i = buf_i \cap N(v'_i, l)$;
14 do prefix-sum scan on $\{num(buf_i)\}$;
15 allocate memory for new intermediate table M;
16 launch a GPU kernel function to link M and $buf_{0,...|M|-1}$ to generate M ;
17 **forall** *partial answer* m_i *in* M' **do**
18 /*$m_i \times buf_i$*/ ;
19 read m_i into shared memory;
20 assign a unique warp w_i to deal with m_i;
21 **forall** z *in* buf_i **do**
22 copy m_i and z to the corresponding address of M as a new row;
23 return M as the result;

Algorithm 3 shows how to allocate buf_i for each row m_i. Assume that there exist multiple linking edges between Q' (the matched partial query graph) and vertex u (to be joined). To reduce the size of GBA, among all linking edges, we select the linking edge $\overline{u'_0u}$ whose edge label l_0 has the minimum frequency in G (Line 1). We perform a parallel exclusive prefix-sum scan on each row's upper bound $|N(v'_i, l_0)|$ (Lines 3-5), later the offsets ($F[i]$, $\forall 0 \leq i < |M|$) and capacity of GBA ($F[|M|]$) are acquired immediately. With the computed capacity, we pre-allocate the GBA and offset array $F[0, ..., |M| - 1]$ (Line 7). Each buffer buf_i begins with the offset $F[i]$.

Figure 8(a) depicts the process of GBA allocation. First, a parallel exclusive prefix sum is done on $num(N(v_i, a))$ and the size of GBA is computed (200). Then GBA is allocated in global memory and the address of buf_i is acquired. For example, the final row m_{99} has three edges labeled by a, thus $num(N(v_{99}, a))$ is 3 and the beginning address of buf_{99} in GBA is 197. However, if $\overline{u_0u_2}$ is selected as the first edge e_0, we can yield smaller GBA (100). The label b of $\overline{u_0u_2}$ is more infrequent than a, thus heuristically it is superior, as illustrated in Algorithm 3. For ease of presentation, we still assume that $\overline{u_1u_2}$ is selected as e_0.

In each join iteration, Algorithm 2 handles all linking edges between Q' and u. It allocates GBA (Line 1), processes linking edges one by one (Lines 2-13), and finally generates a new intermediate table M (Lines 14-22). Obviously, GBA is allocated only once in Algorithm 2 and no new temporary buffer is needed. Figure 8(a) performs the GBA allocation by edge $\overline{u_1u_2}$ and Figure 8(b) finishes set operations. Correspondingly, edge $\overline{u_1u_2}$ is joined first. For example, $N(v_{99}, a)$ subtracts m_{99} and the result is $\{v_{200}, v_{201}\}$, which are stored in buf_{99} (Line 10). Next, for each valid element x in buf_{99}, we check its existence in candidate set of u_2 (Line 11). The second edge is $\overline{u_0u_2}$ and it is processed by Line 13, where buf_{99} is further intersected with L^b_{99} and the result is $\{v_{201}\}$, i.e., $num(buf_{99}) = 1$. We acquire the matching vertices of each row m_i in buf_i, then a new prefix sum is performed to obtain size and offsets of M (Line 14). After M is allocated, w_i copies extensions of m_i to M (Lines 15-22).

Algorithm 3: Function: Pre-allocate Memory

Input: query graph Q, current intermediate table M corresponding to the partial matched query Q', candidate set $C(u)$ (u is the query vertex to be joined), and linking edges ES between Q' and u.
Output: Allocated memory GBA and Offset arrary F.

1 Among all edges in ES, select edge $e_0 = \overline{u'_0u}$, whose edge label l_0 has the minimum frequency in G.
2 Set offset $F[0]$=0;
3 **foreach** *row* m_i *in* M, $i = 0, ..., |M| - 1$ **do**
4 Assume vertex v'_i matches query vertex u'_0 in row m_i.
5 $F[i+1]$=$F(i)$+$|N(v'_i, l_0)|$. // Do exclusive prefix-sum scan.
6 Let $|GBA| = F[|M|]$;
7 Allocate consecutive memory with size $|GBA|$ and let GBA record the beginning address.
8 Return GBA and offset array $F[0, ..., |M| - 1]$.

GPU-friendly Set Operation. In Algorithm 2, set operations (Lines 10,11,13) are in the innermost loop, thus frequently performed. Traditional methods (e.g., [46]) target the intersection of two lists. However, in our case there are many lists of different granularity for set operations. A naive implementation launches a new kernel function for each set operation and uses traditional methods to solve it. This method performs inefficiently due to load imbalance, so we propose a new GPU-friendly solution to fit set interesection with different granularities.

We use one warp for each row and design different strategies for different granularities: small (partial match M_i), medium (neighbor list $N(v, l)$) and large (candidate set $C(u)$):

- For small list M_i, we cache it on shared memory until the subtraction finishes.
- For medium list $N(v, l)$, we read it batch-by-batch (each *batch* is 128B) and cache it in shared memory, to minimize memory transactions.
- For large list $C(u)$, we first transform it into a bitset, then use exactly one memory transaction to check if vertex v belongs to $C(u)$.

Lines 10 and 11 can be combined together. After subtraction, the check in Line 11 is performed on the fly.

We also add a write cache to save write transactions, as there are enormous invalid intermediate results that do not need to be written back to buf_i. It is exactly 128B for each warp and implemented by shared memory. Valid elements are added to

cache first instead of written to global memory directly. Only when it is full, the warp flushes its cached content to global memory using exactly one memory transaction.

6 Scaling to Large Graphs

In real-world applications, graphs may be too large (e.g., several billions) to be resident in GPU's global memory. In this case, CPU-GPU hybrid computing provides a more scalable solution. In these environments, it is necessary to partition both the graph and intermediate tables as well as the auxiliary data structures we have proposed in previous sections. We first discuss how to partition the signature table as well as PCSR (representing data graphs) in Section 6.1. The join processing over the partition of intermediate result tables is discussed in Section 6.2.

Algorithm 4: Partitioned join of the first edge

Input: intermediate table M', candidate set $C(u)$, linking edge $e = \overline{u'u}$ with edge label l and partitioned PCSRs for all segments of l-partitioned graphs.
Output: matches of u for each row of M'

```
1  Initialize flag_i := 0 for each row m_i
2  forall segment x do
3      transfer x to GPU memory ;
4      forall each row m_i (partial match) in M' do
5          /*Each warp handles each row m_i in parallel*/
6          assume that v'_i match u' in m_i;
7          if flag_i < 0 then
8              continue ;
9          if v'_i exists in x then
10             read v'_i's neighbors N_x(v', l) in x;
11             append N_x(v', l) \ m_i ∩ C(u) to buf_i ;
12             set flag_i := 1 if flag_i = 0 ;
13         else
14             set flag_i := −1 if flag_i > 0 ;
15 return buf_i for each row m_i of M';
```

6.1 Partition of graph structures

Signature Table Partition. As discussed in Section 3.1, we use the vertex signature table to filter out unpromising vertices. In large data graphs, the signature table may itself be beyond the capacity of GPU memory. Furthermore, a longer signature is needed to improve the pruning power over large graphs, which further aggravates the space overhead. We vertically divide a large signature table that cannot be resident in GPU memory into several buckets, as shown in Figure 11(b). Each bucket contains the same number of columns, as the signature table is stored in column-first layout. If a bucket is still too large (even if there is only one column in a given bucket), we further divide each bucket horizontally into different *segments* such that (1) each segment contains the same number of vertices (called *vertex subset*)(Figure 11(c)), and (2) the size of each segment fits GPU memory. For each vertex subset vs, all of its corresponding buckets are processed iteratively, after which the valid results from all vertex subsets are unioned. Note that the number of vertices in each segment should be multiples of 32, which favors the coalesce memory accesses in a warp (a warp has 32 threads).

PCSR Partition. For large data graphs, PCSR structures become too large to be placed in GPU memory, requiring their partitioning. Recall that, in building the PCSR, the original data graph has been divided into different edge label-partitioned graphs. Note that each PCSR corresponds to a label-partitioned graph. Each

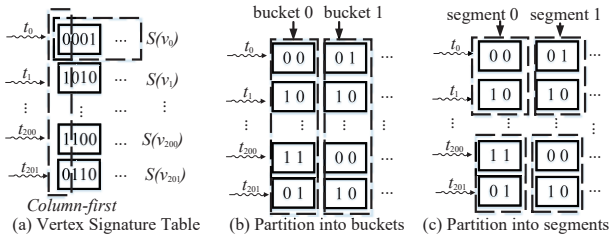


Fig. 11. Partition of the signature table

time we consider a linking edge (Lines 2-13 in Algorithm 2), only the corresponding PCSR needs to be placed in GPU. However, for some frequent edge labels, such as rdf:type in RDF graphs, the corresponding PCSR may not fit into GPU memory. The solution is to further divide a large edge label-partitioned graph into segments and to build a small-size PCSR for each segment. A straightforward strategy is to divide vertices into multiple segments, and all edges incident to vertex v are placed in v's segment. However, the vertex-oriented segmentation may lead to workload imbalance, since the vertex degree distribution is often skewed (e.g., $N(v_0, l)$ has only one neighbor, but $N(v'_i, l)$ has eight neighbors in Figure 12(a)). Thus, to achieve better balance, edge-oriented partitioning is more desirable. Specifically, we sort all edges (in a label-partitioned graph) by source and destination nodes. Then, we evenly divide these edges into different segments (e.g., Figure 12(a)). These segments are also ranked by segment IDs. Obviously, the incident edges of a vertex (i.e., $N(v, l)$) may be assigned to several continuous segments. Thus, extracting one vertex v's neighbors based on edge label l (i.e.$N(v, l)$) may need to be performed over several consecutive segments if $N(v, l)$ is large (e.g., $N(v'_i, l)$ spans three segments in Figure 12(a)). All segments share the same number of edges and the space cost of each segment is within the capacity of GPU memory. This edge-oriented partitioning is much better if the data graph has highly skewed degree distribution.

Algorithm 5: Partitioned join of non-first edge

Input: intermediate table M', linking edge $e = \overline{u'u}$ and all graph structure segments corresponding to the label l of e.
Output: matches of u for each row of M'

```
1  Initialize flag_i := 0 for each row m_i
2  forall segment x do
3      transfer x to GPU memory ;
4      forall each row m_i (partial match) in M do
5          /*Each warp handles each row m_i in parallel*/
6          assume that v'_i match u' in m_i;
7          if flag_i < 0 then
8              continue ;
9          if v'_i exists in x then
10             read v'_i's neighbors N_x(v'_i, l)in x;
11             let buf_i[z + 1] be the first element (in
                 buf_i[flag_i :]) exceeding the last element of vns ;
12             merge intersection of buf_i[flag_i : z] with
                 N_x(v'_i, l);
13             set flag_i := z + 1 ;
14             set flag_i := −1 if flag_i ≥ |buf_i| ;
15         else
16             set flag_i := −1 if flag_i > 0 ;
17 return buf_i for each row m_i of M;
```

Set Operations in Join. In Algorithm 2, Lines 10-11 and 13

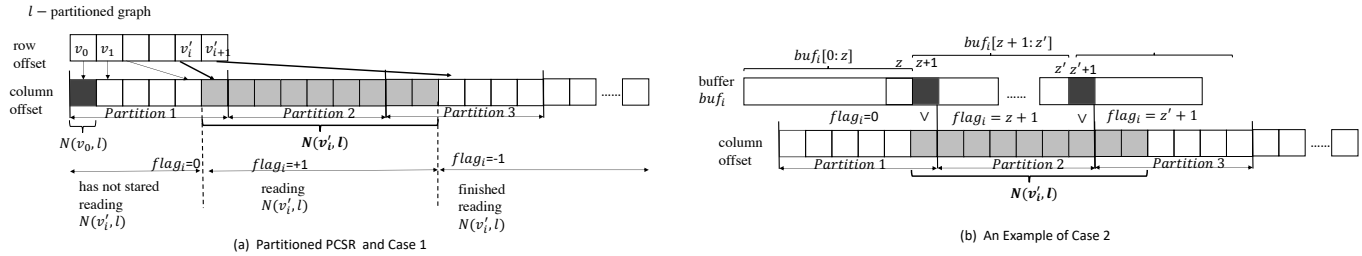


Fig. 12. Partition of the graph structure

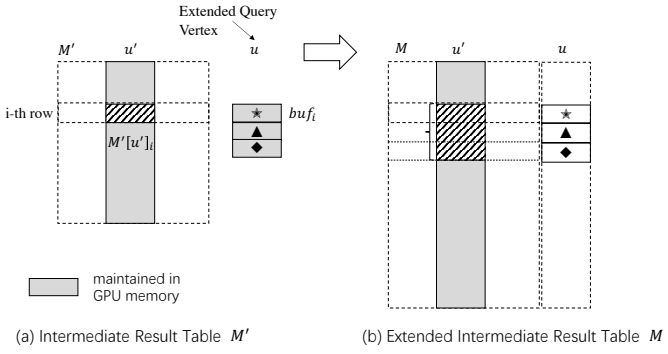


Fig. 13. Extending Partitioned Intermediate results

Algorithm 6: Generating New Intermediate Table

Input: subquery Q' and extended subquery $Q = Q' \cup u$, intermediate table M' and buffer buf_i
Output: new intermediate table M

```
1  forall each column M'[u'] in the intermediate table M' do
2      Load M'[u'] into GPU memory
3      forall each i-th row M'[u']_i in M'[u'] do
4          if buffer buf_i = φ then
5              continue ;
6          duplicate M'[u']_i by num(buf_i) times into M[u']
7      flush M[u'] to host memory and empty the whole GPU memory
8  forall each buffer buf_i do
9      forall each valid item z in buf_i do
10         copy z into M[u]
11 flush M[u] to host memory
12 return the new intermediate table M
```

(set operations) need to be changed to adapt to the partitioned PCSR, since one vertex's neighbors may be located at multiple consecutive segments (i.e., Algorithms 4 and 5 in Appendix B of the supplementary). For each linking edge e with label l, all segments of the l-partitioned subgraph need to be considered (Line 2 in Algorithms 4 and 5). We sequentially scan these segments. For each row m_i (partial match) in the intermediate result table M', we need to check if it can be extended to a larger match by probing m_i's neighbors. Each warp handles each row m_i independently and is assigned a buffer buf_i to store extended vertices. Due to the partitioned PCSR, we introduce a flag variable $flag_i$ for each row m_i. Initially, $flag_i = 0$. We consider two cases for the linking edge e. In Algorithms 4 and 5, we consider the linking edge $e = \overline{u'u}$, v'_i matches u' in each row m_i, and u is the vertex to be joined. Figure 12 illustrates the two cases.

Case 1: $e = \overline{u'u}$ is the first edge (Algorithm 4)

The variable $flag_i$ has three states:

1) $flag_i = 0$: Reading of v_i's neighbors $N(v'_i, l)$ has not yet started (Lines 7-8 in Algorithm 4);
2) $flag_i = +1$:Reading of neighbors $N(v'_i, l)$ has started, but has not yet finished (Lines 9-12);
3) $flag_i = -1$: Reading of $N(v'_i, l)$ is complete (Lines 13-14).

Initially, $buf_i = \phi$. We need to compute $buf_i = (N(v'_i, l) \setminus m_i) \cap C(u)$ by Algorithm 4. Since PCSR is partitioned, $N(v'_i, l) = \bigcup_{x=1}^{n} N_x(v'_i, l)$, where $N_x(v'_i, l)$ is v_i's neighbors in segment x. Thus, $buf_i = \bigcup_{x=1}^{n}(N_x(v'_i, l) \setminus m_i \cap C(u))$. We sequentially scan these segments according to l-partitioned graphs and process each one. For a segment x, if $flag_i \geq 0$, we need to check if v'_i exists in x. If it does, we compute $(N_x(v'_i, l) \setminus m_i) \cap C(u)$ and update the flag $flag_i = +1$ if it is 0. If v'_i is not in x and $flag_i = +1$, wet set the flag to -1, and the corresponding warp of m_i will ignore the rest of the segments.

Case 2: $e = \overline{u'u}$ is not the first edge (Algorithm 5)

Variable $flag_i$ has the same first and last states as in Case 1, but the second state is different:

2) $flag_i > 0$: Reading of neighbors $N(v, l)$ has started, and $flag_i$ denotes the start position (index) of the next merge intersection (Lines 9-14 in Algorithm 5);

As in Case 2, $N(v'_i, l) = \bigcup_{x=1}^{n} N_x(v'_i, l)$ and $buf_i \cap N(v'_i, l) = \bigcup_{x=1}^{n}(buf_i \cap N_x(v'_i, l))$ (Line 12 in Algorithm 5).

Since all vertex IDs in each $N_x(v'_i, l)$ and in buf_i are sorted and we process each $N_x(v'_i, l)$ sequentially, we can reduce the search space for each $buf_i \cap N_x(v'_i, l)$ computation as follows. If v'_i exists in current segment x (Line 9 in Algorithm 5), we read the neighbors $N_x(v'_i, l)$ (Line 10). Let $buf_i[z+1]$ $(z + 1 \geq flag_i)$[3] be the first element larger than the last element in $N_x(v'_i, l)$. Obviously, we only need to merge $buf_i[flag_i : z]$ (the elements from position $flag_i$ to z in buf_i) with $N_x(v'_i, l)$ (Line 10). Note that $flag_i$ is initialized as 0. We update $flag_i = z + 1$ to process the next segment (Line 13). If v'_i does not exist in segment x and $flag_i > 0$, we set $flag_i = -1$ and ignore the rest segments for m_i (Line 7).

In both cases, if $flag_i = -1$, the corresponding warp of m_i will ignore the rest of the segments. Furthermore, if for all matches m_i in the intermediate result table M, the corresponding $flag_i = -1$, it means that we can *early terminate* scanning the rest of the segments.

6.2 Intermediate results and join processing

The challenge of dealing with large graphs is not only the size of data graph but also the unbounded intermediate re-

3. z can be found by binary search; it can also be implicitly acquired in the end of merge intersection.

sult sizes. It is known that, in the worst case, there may be $|V(G)|^{|V(Q)|}$ matches with a maximum space requirement of $O(|V(G)|^{|V(Q)|} \times |V(Q)|)$. Obviously, this is not feasible for the GPU memory when the data graph is large. We propose several ways to partition the intermediate tables to fit GPU global memory. Figure 13 illustrates a running example. Note that, for notational consistency, we use M' to denote the intermediate table and M to denote the new extended table (same as Algorithm 2).

Recall that for each row m_i in M', if buffer buf_i is not empty, $\{m_i\} \times buf_i$ produces new matches (in the new extended table M) of the extended subquery (Lines 18 to 22 in Algorithm 2). Each row m_i is processed in parallel. However, Algorithm 2 requires that both the intermediate table M' and the new extended table M should be cached in GPU memory. When they are too large to be resident in GPU memory, we store all intermediate tables (including final result table) in the columnar fashion.

In each join iteration, we only process one linking edge $\overline{u'u}$ (Lines 2-13 in Algorithm 2), i.e., only column $M[u']$ is needed when processing linking edge $\overline{u'u}$ (as in Figure 13). Thus, we keep only the needed column of $M'[u']$ in GPU memory in each iteration (Lines 2-13). This saves memory and memory accesses are well coalesced with the column-oriented storage of M'.

After processing all linking edges in ES (after Lines 13 in Algorithm 2), according to valid items in buffer buf_i, we build the new extended table M in Algorithm 6 (see Appendix B of the supplementary material). Figure 13 illustrates how to extend the intermediate results in a columnar fashion. Computation iterates over columns in the original intermediate table M' (Lines 1-7 in Algorithm 6). When processing column $M'[u']$, we consider each of its rows indexed by i. If $buf_i = \phi$, it means that the current match of Q' cannot be extended to match extended query $Q = Q' \cup u$, and we ignore this row (Lines 4-5). Otherwise, we duplicate $M'[u']_i$ by $num(buf_i)$ times into $M[u']$ (Line 6), where $num(buf_i)$ denotes the number of valid items in buffer buf_i, and $M[u']$ denotes the corresponding column in the new extended table M. Finally, we copy valid items in buffer buf_i into the new added column $M[u]$ (Lines 8-11).

Algorithm 6 maintains at most two columns $M'[u']$ and $M[u']$ in GPU memory (demonstrated in shaded areas of Figure 13), which is a significant improvement over Algorithm 2. If the two columns are still too large to be resident in GPU memory, we can further partition each column into several subtables horizontally and employ a pipeline strategy to process subtables.

7 Load Balancing

Algorithm 2 processes rows in the intermediate table in parallel. However, since the vertex degree of many real graphs follows power-law distribution, it will lead to serious workload imbalance when each warp deals with each row (Line 7). As mentioned in Section 2.2, GPU employs the 4-layer thread hierarchy (kernels, blocks, warps, and threads) that can be utilized to develop a *4-layer balance scheme* to address workload imbalance.

In this section, we study how to set up a 4-layer balance scheme by a quantitative cost model. For convenience, we measure the workload size by neighborhood set size, i.e., $N(v_{i'}, l)$ in Line 10. According to neighborhood set size, we utilize different thread hierarchy to process each row in parallel. Let us see Figure 18(a). We introduce three parameters ($W_3 < W_2 < W_1$) to select different thread hierarchy for different workloads.

1) Extract rows whose workload sizes ($N(v_{i'}, l)$) exceed W_1, and dynamically launch a new kernel function to handle each one;
2) Each row whose workload size is within $(W_2, W_1]$ is processed by one whole block.
3) Load rows whose workload size are within $(W_3, W_2]$ to shared memory and divide them evenly by all warps in a block. We use multiple warps (in a block) to handle one row when its workload size is within $(W_3, W_2]$.
4) Each remaining task of the corresponding row (workload size is below W_3) is processed by one warp.

We set $W_2 = 1024$, the maximum number of threads in a block, to gain more opportunities of scheduling within a block. The 4-layer balance scheme is superior to merging all tasks and dividing them equally [47], because it reduces the overhead of merging tasks into the work pool. Our experiments (see Table 12 of Appendix D) also confirm that.

The key issue in the 4-layer balance scheme is how to set parameter values for W_i. An empirical method tunes these parameters according to real performance comparison [48]. Obviously, the parameter tuning is a long and tedious process. Thus, we propose a histogram-based quantitative method to solve this problem.

Let one task process each row in Algorithm 2. Assume that we have n tasks $T_1, T_2, \ldots, T_n$. A histogram is built to collect task statistics, where each interval is $I_k = [32 \times k, 32 \times (k+1))$. The first interval of this histogram is [0, 32), while the last interval is $I_b = [32 \times b, 32 \times (b+1))$ (here $b = \lfloor \frac{\max\{|T_i|\}}{32} \rfloor$). Each task T_i falls into some interval I_k if $32 \times k \leq A_i < 32 \times (k+1)$. To generate a histogram, for each interval I_k, we accumulate the number of tasks that fall into I_k denoted as $|I_k|$. The histogram generation can be implemented quickly on GPU, exploiting parallelism [49]. We also adopt the 4-layer balance scheme by dividing the histograms into four partitions. An example of histogram-based balance scheme is given in Figure 18(b) of Appendix B.

We design a quantitative cost model based on GPU architecture to set up parameters. Let $|SMs|$ be the number of SMs available on the GPU. Each SM can run a block (1024 threads) and each block consists of 32 warps.

First, consider the cost of the leftmost intervals, i.e., the neighborhood size is less than W_3. If an interval I_i is smaller than W_3 (see Figure 18(b) of Appendix B), each task is processed by a warp. The workload size of one task in interval I_i is estimated as $32 \times i$. Since one warp has 32 parallel threads, thus, we define the cost of each task (in interval I_i on the left of W_3) is $\frac{32 \times i}{32} = i$. Note that I_i contains $|I_i|$ tasks and a GPU has $32 \times |SMs|$ warps in total. Therefore, the total computation cost of I_i can be estimated by

$$f_{a-warp}(i) = i \times \lceil \frac{|I_i|}{32|SMs|} \rceil$$

Second, consider the intervals (I_i) between W_3 and W_2. Tasks belonging to I_i are processed by multiple warps that divide these tasks evenly. Tasks are added to a task pool (size 1024) in shared memory, then all threads of a block process them iteratively. For example, assuming three tasks (R_1, R_2 and R_3) have 300, 400 and 500 elements respectively, they will be added to the task pool and divided evenly over 1024 threads, each thread processing one element. In the first iteration, the task pool consists of 300 elements of R_1, 400 elements of R_2 and 324 elements of R_3. After these 1024 elements are processed, in the second iteration,

the remaining 176 elements of R_3 will be added into the task pool and processed. In each iteration, two synchronization calls are needed within the block: one for merging tasks, the other for processing elements. The cost of a synchronization call can be estimated by experiments, and we denote it as t_{sn}. Each SM is assigned $\lceil \frac{|I_i|}{|SMs|} \rceil$ tasks. According to histogram estimation, if a task is in I_i, the workload size is estimated as $32 \times i$. Therefore, the total work amount of each SM is $32i \times \lceil \frac{|I_i|}{|SMs|} \rceil$. Furthermore, a SM has 1024 threads, thus, the cost of processing all tasks in I_i is

$$f_{m-warp}(i) = (1 + 2t_{sn}) \times \lceil \frac{32i \times \lceil \frac{|I_i|}{|SMs|} \rceil}{1024} \rceil$$

Third, consider the intervals (I_i) between W_2 and W_1. Each task of I_i is processed by a block. Each iteration also needs a synchronization call within the block. Each SM processes $\lceil \frac{|I_i|}{|SMs|} \rceil$ tasks, and each task is processed by a block independently. Thus the cost is

$$f_{a-block}(i) = (1 + t_{sn}) \times \lceil \frac{|I_i|}{|SMs|} \rceil \times \lceil \frac{32i}{1024} \rceil$$

Finally, consider the rightmost intervals (i.e., larger than W_1). Let t_{kl} be the cost of launching a kernel function. The real value of t_{kl} can be estimated offline by launching many lightweight kernel functions on GPU and recording the average. If I_i is on the right of W_1, the cost will be

$$f_{kernel}(i) = |I_i| \times (t_{kl} + \lceil \frac{32i}{1024 \times |SMs|} \rceil)$$

In the histogram, we set the interval width be 32. For convenience, we set $W_1 = 32x_1$, $W_2 = 32 \times 32 = 1024$ and $W_3 = 32x_3$, where $0 \le x_3 \le x_1 \le b$, $x_1 \ge 32$ and $b = \lfloor \frac{\max\{|T_i|\}}{32} \rfloor$. The entire histogram can be divided into four parts. The whole computing cost is modeled in Equation 1.

$$\begin{aligned} C = &\textstyle\sum_{i=0}^{i=x_3} f_{a-warp}(i) + \sum_{i=x_3+1}^{i=31} f_{m-warp}(i) \\ &+ \textstyle\sum_{i=32}^{i=x_1} f_{a-block}(i) + \sum_{i=x_1+1}^{i=b} f_{kernel}(i) \end{aligned} \quad (1)$$

Our goal is to minimize the above equation by tuning parameters x_1 and x_3. This optimization problem can be solved quickly on a GPU even using brute-force enumeration. Since $0 \le x_3 \le x_1 \le b$, the number of enumerations is upper bounded by $(b+1)^2$. In our experiments, the parameter setting takes less than one second, which is only $\frac{1}{45}$ of our SGSI algorithm (see Section 8.4). This parameter setting strategy leads to more than 1.6× speedup (see Table 5).

8 EXPERIMENTS

We evaluate our method (called *GSI*) against state-of-the-art subgraph matching algorithmss: CPU-based VF3 [29], CFL-Match [8], CBWJ [13], and GPU-based GpSM [9] and GunrockSM [10]. We also include the two state-of-the-art GPU-based RDF systems MAGiQ [37] and Wukong+G [38]. Note that RDF systems are originally designed for SPARQL queries whose semantics is subgraph homomorphism; we extend them to support subgraph isomorphism. To verify the efficiency of our method in dealing with large graphs (called *SGSI*) that cannot be resident in GPU memory, we also compare with PBE [15] that has recently been proposed for this case. Although PBE targets unlabeled graphs, it is the only work that considers the out-of-core GPU-based solution for subgraph query in the literature. To enable fair comparison, we revise of PBE code to handle labelled graphs, for example by introducing the label constraints when considering subgraph match. We have ensured that both SGSI and PBE produce the same results. All experiments are performed on a workstation running CentOS 7 and equipped with Intel Xeon E5-2697 2.30GHz CPU and 256G main memory, NVIDIA Titan XP with 30 SMs (each SM has 128 cores and 48KB shared memory) and 12GB global memory. Furthermore, to test the scalabitlity of our approach, we perform all experiments on large graphs in Section 8.4 using Nvidia DGX A100 workstation with 6912 cuda cores and 80GB global memory as well as AMD Rome 7742 with 64 cores.

TABLE 2
Statistics of Datasets

Name	$\|V\|$	$\|E\|$	$\|L_V\|$	$\|L_E\|$	MD[1]	Type[2]
enron	69K	274K	10	100	1.7K	rs
gowalla	196K	1.9M	100	100	29K	rs
patent	6M	16M	453	1K	793	rs
road	14M	16M	1K	1K	8	rm
DBpedia	22M	170M	1K	57K	2.2M	rs
WatDiv	10M	109M	1K	86	671K	s
DB	233M	1.1B	1K	124K	17M	rs
FR	65M	1.8B	1K	1K	5.2K	rs
YH	417M	2.8B	1K	1K	2.5K	rs
WD	97M	1B	1K	86	6.7M	s

* $|L_V|$ and $|L_E|$ denote the number of vertex label and edge label, respectively.
[1] Maximum degree of the graph.
[2] Graph type: r:real-world, s:scale-free, and m:mesh-like.

8.1 Datasets and Queries

The experiments are conducted on both real and synthetic datasets listed in Table 2. We further use four large graphs that exceed the capacity of GPU memory to evaluate SGSI: DBpedia1B (DB), friendster network (FR), Yahoo network (YH) and WatDiv1B (WD). Since most graphs do not contain vertex/edge labels (except for edge labels in RDF datasets and vertex labels in patent dataset), we assign labels following the power-low distribution. The default numbers of vertex/edge labels are given in Table 2. To generate a query graph, we follow the same query graph generation approaches in previous studies [50], [51] where a random walk is performed over the data graph G starting from a randomly selected vertex until $|V(Q)|$ vertices are visited. All visited vertices and edges (including the labels) form a query graph.

For each query size $|V(Q)|$, we generate 100 query graphs and report the average query time. The default query size $|V(Q)|$ is 12 in the following experiments. We also evaluate GSI with respect to the number of vertex/edge labels and query size.

8.2 Evaluating Join Phase

We evaluate three componenets that contribute to the performance of GSI's join phase: PCSR structure, the Prealloc-Combine strategy and GPU-friendly set operation. Table 3 shows the results, where GSI-b is the basic implementation with traditional CSR structure, two-step output scheme and naive set operation. Two metrics are compared: (1) the number of transactions for reading data from global memory (GLD), and (2) the query response time of answering subgraph search query. We add these techniques to GSI-b one-at-a-time, and compare the performance of each technique with previous implementation. For example, in Table

TABLE 3
Performance of techniques in join phase

Dataset	Global Memory Load Transactions (GLD)							Query Response Time (ms)						
	GSI-b[1]	+DS[2]	drop[3]	+PC[4]	drop	+SO[5]	drop	GSI-b	+DS	speedup	+PC	speedup	+SO	speedup
enron	3M	2.1M	30%	1.6M	25%	656K	59%	573	274	2.1×	176	1.6×	28	6.3×
gowalla	3.2M	2M	38%	1.3M	33%	848K	39%	353	172	2.1×	88	2.0×	69	1.3×
patent	3.5M	2.4M	31%	1.8M	25%	1.6M	11%	3K	1.4K	2.1×	700	2.0×	524	1.3×
road	3.4M	2.2M	35%	1.7M	22%	1.6M	5%	2.4K	675	3.6×	456	1.5×	456	1.0×
WatDiv	40M	30M	25%	21M	28%	13M	39%	43K	31K	1.4×	25K	1.2×	1K	25×
DBpedia	53M	31M	42%	24M	21%	14M	43%	85K	48K	1.8×	36K	1.3×	2K	18×

[1] Basic GSI implementation with traditional CSR structure, two-step scheme and naive set operation.
[3] drop: the drop of the number of transactions for reading data from global memory (GLD).
[2,4,5] Add techniques to GSI- one by one: PCSR structure, Prealloc-Combine strategy and GPU-friendly set operation.

3, the column "+SO" is compared with the column "+PC" to compute GLD drop and speedup. GSI incorporates all techniques.

TABLE 4
Comparison of CR and PCSR

Dataset	GLD			Time (ms)		
	CR	PCSR	drop	CR	PCSR	speedup
enron	2.4M	2.1M	13%	311	274	1.1×
gowalla	2.5M	2M	20%	212	172	1.2×
patent	3.2M	2.4M	25%	1.8K	1.4K	1.3×
road	3.0M	2.2M	27%	873	675	1.3×
WatDiv	46M	30M	35%	42K	31K	1.4×
DBpedia	37M	31M	16%	56K	48K	1.2×

8.2.1 Performance of PCSR structure

To verify the efficiency of PCSR, we compare it with traditional CSR structure. We set the bucket size to 128B and find that the maximum length of conflict list is below 15, even on the largest dataset. Therefore, with PCSR structure, GSI always finds the address of $N(v, l)$ within one memory transaction, which is a big improvement compared to traditional CSR.

Table 3 shows that PCSR brings an observable drop of GLD (about 30%), and nearly 2.0× speedup. The least improvement is observed on WatDiv due to small $|L_E|$, while on other datasets the power of PCSR is tremendous, achieving more than 1.8× speedup. The superiority of PCSR is mainly due to two factors: (1) fewer memory transactions are needed, as presented in Table 8, and (2) threads are fully utilized while traditional CSR suffers heavily from thread underutilization.

We also compare PCSR with "Compressed Representation" (CR) in Table 4. On all datasets, at least 13% drop of GLD and 1.1× speedup are achieved. The best performance is observed over WatDiv, where CR has even higher GLD than traditional CSR. The reason is that CR is much more complex when locating the position of $N(v, l)$. WatDiv has the minimum number of edge labels, thus its edge label-partitioned graphs are very large, leading to high cost of locating for CR. The "Basic Representation" (BR) consumes too much memory to run on large graphs with hundreds of edge labels.

8.2.2 Performance of Parallel Join Algorithm

In our vertex-oriented join strategy, there are two main parts: the Prealloc-Combine strategy (PC) and GPU-friendly set operation (SO). To evaluate Prealloc-Combine strategy, we implement the join-twice output scheme [9] as the baseline. Table 3 shows that on all datasets, PC obtains more than 21% drop of GLD and 1.2× speedup. The gain originates from the elimination of double work during join, which also helps reduce GLD, thus further boosts the performance. PC can reduce the amount of work by at most half, thus there is no speedup larger than 2.0×.

To evaluate the proposed GPU-friendly set operation, we compare with naive solution: finish each set operation with a new kernel function. Table 3 shows that SO reduces GLD by about 40%; consequently, it leads to more than 1.3× speed up. On patent and road datasets, the improvement is not apparent because their neighbor lists are relatively small. SO also eliminates the cost of launching many kernel functions.

SO performs best on enron, WatDiv and DBpedia, showing $> 18\times$ speedup. The reason is that write cache performs best on these graphs, thus saving lots of global memory store transactions (GST). On other graphs, the gain of write cache is small because they have fewer matches, thus perform fewer write operations.

8.3 Comparison of GSI with counterparts

Overall Performance. The results are given in Figure 14. Note that there is no bar if the corresponding time exceeds the threshold of 100 seconds. In all experiments, GPU solutions beat CPU solutions as expected due to the power of massive parallelism.

There is no clear winner between the four existing GPU solutions, but none of them are competitive with GSI. GSI answers queries within one second in the first four datasets. On WatDiv and DBpedia, GSI achieves more than 4× speedup over counterparts. Generally, GSI outperforms existing systems on all datasets by several orders of magnitude.

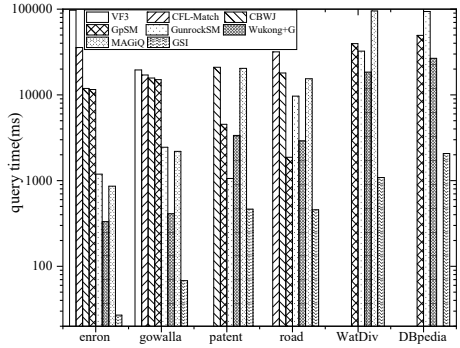
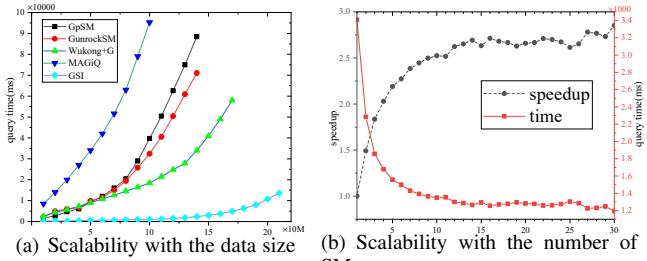


Fig. 14. Performance Comparison on all datasets

Scalability. To evaluate GSI's scalability, we generate a series of RDF datasets using the WatDiv benchmark, varying the number of edges from 10 million to 210 million. CPU solutions fail to run even on the smallest size graph with 10 million edges; thus, we only compare GSI with existing GPU solutions in Figure 15(a). Note that we ignore some algorithm if its running time exceeds

100s, or it runs out of GPU memory. Figure 15(a) shows that the running time of existing GPU solutions grows much faster than GSI, because they have larger candidate tables and intermediate tables. Due to the efficient filter, GSI occupies less memory, thus scales to larger graphs as long as they can fit GPU memory.



(a) Scalability with the data size (b) Scalability with the number of SMs

Fig. 15. Scalability Test

We also evaluate the scalability of GSI with the number of SMs in GPU. In order to control the number of running SMs, we limit the number of blocks launched and withdraw dynamic parallelism, which may degrade the performance.We choose WatDiv (see Table 2) and show the result in Figure 15(b). With the number of SMs increasing from 1 to 30, the response time drops continuously, though with some tiny fluctuations. The time curve drops fast in the beginning, but slows down gradually, corresponding to the sub-linear speedup curve. The maximum speedup is 2.85, and is limited by the irregularity of graphs and GPU memory bandwidth, which cause severe load imbalance and high memory latency.

TABLE 5
Elapsed time (s) of new algorithms on large graphs with A100

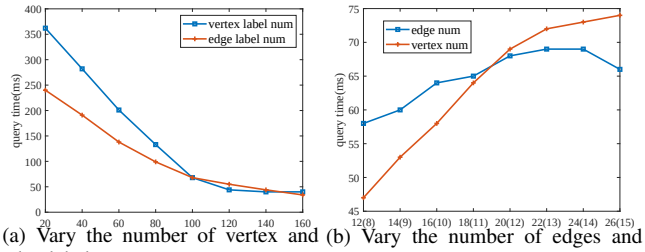
Dataset	PBE	SGSI	speedup	SGSI+	speedup
DB	230	56	4.1×	21	10.9×
FR	390	210	1.8×	148	2.6×
YH	2,123	596	3.5×	403	5.2×
WD	954	282	3.3×	117	8.1×

* All speedups are calculated based on PBE.

Varying the number of labels and query size. We study the performance with regard to the number of edge/vertex labels and query size. We use GSI and select gowalla as the benchmark. By default, the number of vertex and edge labels are both 100, and all queries have 12 vertices. We vary the number of labels and show results in Figure 16(a). As the number of labels increases, run time decreases. The "vertex label num" line shows sharper drop because larger $|L_V|$ directly reduces the sizes of candidate sets. However, after $|L_V| > 100$, the drop quickly slows down to zero as candidate sets are small enough to be fully parallelized. Similarly, larger $|L_E|$ also helps reduce $|C(u)|$ due to improved pruning power of labeled edges. In addition, the size of $|N(v, l)|$ is also lowered as $|L_E|$ grows. This is the reason that run time keeps dropping, though the speed also changes after $|L_E| > 100$.

Loading Time and memory consumption. We also report the loading time (from host to GPU) of GSI on all datasets: 1ms, 5ms, 106ms, 120ms, 144ms, 178ms. We also record the maximum memory consumption of GPU algorithms in Table 6, including host and GPU memories. Note that "NAN" means an algorithm cannot end in a reasonable time. Generally, CPU solutions occupy less memory (see Table 6).

GPU solutions that are based on BFS have larger memory consumption on both host and GPU. Compared to counterparts, GSI consumes more host memory due to the maintainance of signature table and PCSR structures. However, GSI consumes less GPU memory because it has smaller candidate/intermediate tables and in each iteration only an edge label-partitioned graph is needed on GPU.



(a) Vary the number of vertex and edge labels (b) Vary the number of edges and vertices in Q

Fig. 16. Experiments of label number and query size

8.4 Evaluating SGSI on large graphs

In this subsection, we compare SGSI with PBE [15] on four very large graphs that exceed the capacity of GPU memory.

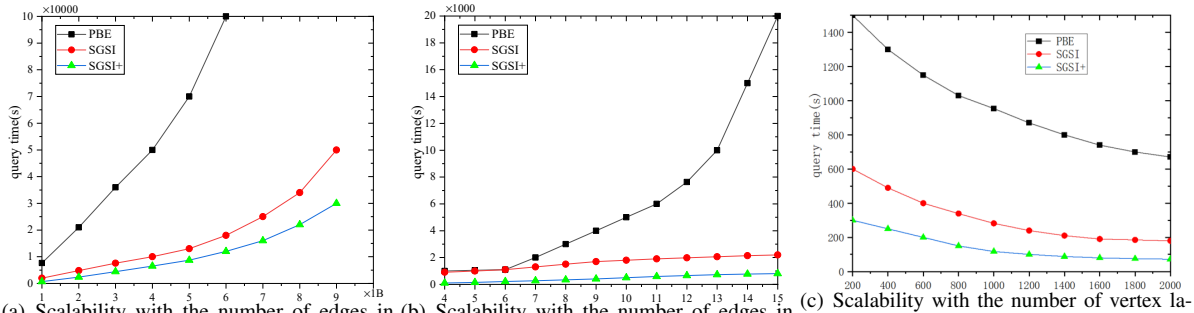
Effects of techniques on large graphs. Table 5 shows that SGSI achieves a high speedup (>1.9×) when compared to PBE on all datasets. This owes to the enhanced design of SGSI, which divides the intermediate table directly rather than search for matches across different graph partitions. The routine of finding cross-matches is a heavy burden for PBE, especially on large queries. It consists of the exploration of all permutations of query edges, which gives rise to much duplicate work. Besides, PBE is not optimized for edge-labeled graphs, while SGSI optimizes the structure and algorithm according to edge labels. Therefore, SGSI achieves the highest speedup (6×) on DB, which has 124K different edge labels. On the largest dataset (YH), SGSI also shows considerable improvement (4× speedup). Generally, queries have many matches on YH, thus the intermediate table is very large and join phase dominates the total time cost. For SGSI, only two columns reside in GPU memory, while PBE has to place the entire intermediate tables (including old table and new table) in GPU memory. Therefore, SGSI shows much better performance on large graphs.

Furthermore, the comparison of the execution time breakdown of PBE and SGSI is reported in Table 7. Obviously, the ratio of $\frac{comm}{comp+comm}$ of PBE is much larger than that of SGSI. The reason is two folds. First, the communication cost of PBE includes two parts. One is due to the division of a large intermediate table beyond the GPU memory capability into small pieces that needs to be accessed. The other one is to search of cross-partition matches. Furthermore, the size of intermediate table of PBE is much larger than that of SGSI, which causes high communication cost of PBE. On the other hand, the memory access is rather distributed for PBE when fetching adjacency lists from main memory while SGSI always reads from main memory in fixed-size blocks sequentially. The distributed memory access scheme brings much more memory transactions, which marks PBE inefficient.

Effects of histogram-based load balance. To evaluate the performance of histogram-based load balancing strategy (denoted as SGSI+), we compare it against the original load balance strategy (denoted as SGSI) based on empirical parameter tuning. Results in Table 5 confirm that the histogram-based strategy achieves higher than 1.6× speedup on all datasets. On DB and WD, the graph data is more skewed, thus the improvement is more apparent. The

TABLE 6
Memory consumption of GPU algorithms

Dataset	Host Memory					GPU Memory				
	GpSM	GunrockSM	Wukong+G	MAGiQ	GSI	GpSM	GunrockSM	Wukong+G	MAGiQ	GSI
enron	154M	181M	174M	284M	160M	1.3G	1.4G	667M	721M	661M
gowalla	592M	712M	599M	367M	466M	2.4G	2.8G	1.8G	725M	685M
patent	1.0G	1.3G	1.3G	1.5G	663M	3.5G	3.7G	2.1G	1.1G	915M
road	1.8G	2.0G	2.1G	2.2G	1.1G	3.6G	3.6G	2.3G	1.3G	1.2G
WatDiv	4.4G	5.7G	4.7G	6.9G	8.5G	7.3G	7.7G	7.5G	5.6G	4.9G
DBpedia	6.9G	8.2G	8G	NAN	14G	9.0G	9.6G	9.6G	NAN	8.1G



(a) Scalability with the number of edges in data graph (b) Scalability with the number of edges in query graph (c) Scalability with the number of vertex labels in query graph

Fig. 17. Scalability on enormous graphs (in WD graphs)

TABLE 7
Elapsed running times (s) of PBE and SGSI on A100 (in ms)

Dataset	Computation		Communication	
	PBE	SGSI	PBE	SGSI
DB	132	12.9	98	8.1
FR	261	111	129	37
YH	1205	323	918	80
WD	489	76.1	465	40.9

speedup is the highest on WD, because it has the minimum number of edge labels, which produces larger skewed workloads. YH is the largest dataset, which has larger workloads, thus the speedup is more obvious on YH than FR. To sum up, histogram-based load balance strategy works the best when the workloads (during join) are large and skewed. When equipped with this strategy, SGSI+ achieves $> 3.0\times$ speedup when compared with PBE ($15\times$, $3.0\times$, $8\times$, and $11.4\times$ on DB, FR, YH, and WD, respectively). Furthermore, the total time of histogram-based preprocessing is very small (<1s on all datasets).

Scalability on large graphs. To evaluate the scalability of PBE and SGSI, we conduct comparison on WatDiv datsets varying the number of edges from 1 billion to 9 billion. Figure 17 summarizes the experiment results. As the data size grows, the elapsed time of all solutions rise, but SGSI and SGSI+ rise much more slowly than PBE. On larger datasets, matches tend to cross more partitions, which negatively impacts the performance of PBE. Furthermore, the intermediate tables become bigger, thus the time cost of collecting neighbors by CPU in PBE is much larger. On WatDiv6B and larger datasets, PBE fails on most queries because it has not been able to generate a result within 24 hours. In comparison, SGSI does not need to find cross-matches, or collect and compact neighbors on CPU. The structures of SGSI are well devised and partitioned, thus there is no problem of exceeding GPU memory. However, when the number of edges is larger than 9 billion, it exceeds the size of host memory (CPU side), which needs a disk-resident solution. That is beyond the scope of this work. Thus, we only report experiment results up to graphs with 9 billion edges.

To study the impact of query size, we vary the number of query vertices from 4 to 15 and report the elapsed time in Figure 17(c). PBE performs very well on queries with less than 7 vertices, sharing similar performance with SGSI, since small queries tend to be in the same partition. In experiments, most matches are included in one or two partitions when the number of query vertices is less than 7. However, on larger queries, an complete subgraph match crosses several partitions, thus the performance drops sharply in PBE. The reasons are three-fold: First, PBE places all columns of the intermediate table in GPU memory, which needs to be divided into very small subtables for a large query. Second, the number of edge permutations increases linearly when the query size increases. Third, to find cross-matches, PBE needs to collect and compact neighbors for the current intermediate table. This process is performed by CPU, thus is very costly when encountering large intermediate tables. Experimentally, on queries involving more than 7 vertices, most matches cross 3-5 partitions. On queries larger than 12, several matches can cross nearly all partitions, which causes severe performance degradation.

When query size increases, the elapsed time also rises in SGSI and SGSI+, as the number of join iterations increases. However, the rate of increase is very slow, due to the optimizations introduced earlier. During each join iteration, only two columns of the intermediate table reside in GPU memory, no matter how large the query is. Furthermore, SGSI does not need to find cross-matches and all structures are well partitioned. In practice, SGSI can easily process queries with more than 30 vertices.

We also evaluate the performance of our method varying the number of vertex labels in WD graphs in Figure 17(c) and it confirms that SGSI can achieve better performance with the increasing number of vertex labels.

9 Conclusions

In this paper, we propose an efficient GPU-based subgraph isomorphism (GSI) algorithm, which is based on filtering-and-joining framework and optimized for the architecture of modern GPUs. The technique contributions of GSI are two aspects: First, we design a GPU-friendly data structure (PCSR) for edge-labelled graphs, which facilitates one primitive operator (i.e., $N(v, l)$ extraction) in GPU, accessing one vertex's all neighbors based on edge label. Second, we are the first to study a vertex-oriented join algorithm (i.e., worst-case optimal join) in GPU and propose a *Prealloc-Combine* strategy to avoid parallel write conflicts. Furthermore, in order to support larger graphs that are beyond the capability of CPU memories, several extensions are proposed to enhance GSI, called SGSI, including the partition of graph structures, the partitions of the intermediate tables, and the histogram-based load balances. SGSI is scalable to large graphs with billions of edges in a single GPU. We release our implementation together with datasets at https://github.com/pkumod/GSI.

Acknowledgments

This work was supported by NSFC under grant 61932001 and U20A20174. M. Tamer Özsu's work was supported by Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2004, pp. 335–346.

[2] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 16:1–16:45, 2009.

[3] O. Lassila, R. R. Swick *et al.*, "Resource description framework (RDF) model and syntax specification," *W3C Recommendation*, 1998.

[4] L. Zeng and L. Zou, "Redesign of the gStore system," *Frontiers Comput. Sci.*, vol. 12, no. 4, pp. 623–641, 2018.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 18, no. 3, pp. 265–298, 2004.

[7] J. Kim, H. Shin, W. Han, S. Hong, and H. Chafi, "Taming Subgraph Isomorphism forRDF Query Processing," *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1238–1249, 2015.

[8] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2016, pp. 1199–1214.

[9] H. N. Tran, J. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *Proc. 20th Int. Conf. on Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, vol. 9049, 2015, pp. 299–315.

[10] L. Wang, Y. Wang, and J. D. Owens, "Fast parallel subgraph matching on the gpu," *Proc. 25th IEEE Int. Symp. High Performance Distributed Computing*, 2016.

[11] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: new developments in the theory of join algorithms," *SIGMOD Rec.*, vol. 42, no. 4, pp. 5–16, 2013.

[12] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2016.

[13] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1692–1704, 2019.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proc. 17th Int. Conf. on Parall. Arch. and Compilation Tech.*, 2008, pp. 260–269.

[15] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2020, pp. 1067–1082.

[16] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[17] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. 2012 IEEE Int. Symp. on Workload Characterization*, 2012, pp. 141–151.

[18] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[19] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Analy. Machine Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004.

[20] P. Zhao and J. Han, "On graph query optimization in large networks," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 340–351, 2010.

[21] K. Zhu, Y. Zhang, X. Lin, G. Zhu, and W. Wang, "NOVA: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs," in *Proc. 15th Int. Conf. on Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, vol. 5981, 2010, pp. 140–154.

[22] P. Peng, L. Zou, L. Chen, X. Lin, and D. Zhao, "Subgraph search over massive disk resident graphs," in *Proc. 23rd Int. Conf. on Scientific and Statistical Database Management*, ser. Lecture Notes in Computer Science, vol. 6809, 2011, pp. 312–321.

[23] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[24] J. Lee, W. Han, R. Kasperovics, and J. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *PVLDB*, 2012.

[25] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2013, pp. 337–348.

[26] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[27] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[28] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *J. Symb. Comput.*, vol. 60, pp. 94–112, 2014.

[29] V. Carletti, P. Foggia, A. Saggese, and M. Vento, "Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 4, pp. 804–818, 2018.

[30] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms: [extended abstract]," in *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2012, pp. 37–48.

[31] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds., 2020, pp. 1083–1098.

[32] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi, "Efficient subgraph matching using GPUs," *Proc. Australasian Database Conf.*, vol. 8506, pp. 74–85, 2014.

[33] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, 2012.

[34] B. Yang, K. Lu, Y.-h. Gao, X.-p. Wang, and K. Xu, "GPU acceleration of subgraph isomorphism search in large scale graph," *Journal of Central South University*, 2015.

[35] J. Jenkins *et al.*, "Lessons learned from exploring the backtracking paradigm on the GPU," in *Proc. 17th Int. Euro-Par Conf.*, ser. Lecture Notes in Computer Science, vol. 6853, 2011, pp. 425–437.

[36] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proc. 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2016, pp. 11:1–11:12.

[37] F. Jamour *et al.*, "Matrix algebra framework for portable, scalable and efficient query engines for RDF graphs," in *Proc. 14th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.*, 2019, pp. 27:1–27:15.

[38] S. Wang *et al.*, "Fast and concurrent rdf queries using rdma-assisted gpu graph exploration," in *USENIX Annual Tech. Conf.*, 2018, pp. 651–664.

[39] L. Wang and J. D. Owens, "Fast gunrock subgraph matching (GSM) on gpus," *arXiv*, vol. abs/2003.01527, 2020.

[40] Y. Wang, Y. Pan, A. A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 3:1–3:49, 2017.

[41] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: A high-performance subgraph matching system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, pp. 21–37, 2021.
[42] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: a pattern-aware graph mining system," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 2020, pp. 13:1–13:16.
[43] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*. ACM, 2021, pp. 378–391.
[44] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: An efficient and flexible graph mining system on CPU and GPU," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1190–1205, 2020.
[45] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs for graph applications," in *Proc. ACM/IEEE Conf. on Supercomputing*, 2014, pp. 781–792.
[46] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader, "Fast and adaptive list intersections on the GPU," in *Proc. 2018 IEEE High Performance Extreme Comp. Conf.*, 2018, pp. 1–7.
[47] D. Merrill, M. Garland, and A. S. Grimshaw, "Scalable GPU graph traversal," in *Proc. 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.
[48] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: Gpu-friendly subgraph isomorphism," *arXiv*, vol. abs/1906.03420, 2019.
[49] R. Shams, R. Kennedy *et al.*, "Efficient histogram algorithms for nvidia cuda compatible devices," in *Proc. Int. Conf. on Signal Processing and Communications Systems*, 2007, pp. 418–422.
[50] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach." in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2004, pp. 335–346.
[51] W. Han, J. Lee, M. Pham, and J. X. Yu, "igraph: A framework for comparisons of disk-based graph indexing techniques," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 449–459, 2010.
[52] R. Orellana, "https://math.dartmouth.edu/archive/m19w03/public_html/," *Discrete Mathematics in Computer Science*, 2003.

Li Zeng got his doctoral degree from Peking University in 2021. His research interests include graph processing and high performance computing.

Lei Zou is a professor in Wangxuan Institute of Computer Technology of Peking University. He is also a faculty member in National Engineering Laboratory for Big Data Analysis and Applications (Peking University) and the Center for Data Science of Peking University. His research interests include graph databases and semantic data management.

M. Tamer Özsu is a University Professor in David R. Cheriton School of Computer Science at the University of Waterloo. His research addresses data engineering aspects of data science, focusing on large scale data distribution and management of unconventional data (e.g., graphs, RDF, and streams). He is a fellow of Royal Society of Canada, AAAS, ACM and IEEE. He is an elected member of Science Academy, Turkey and a member of Sigma Xi.